Framework for End-to-End Tuning and Regression for a High Performance MPI Library on Modern Supercomputers

Thesis

Presented in Partial Fulfillment of the Requirements for Honors Research Distinction in the College of Engineering of Ohio State University

By

Nick Sarkauskas

Undergraduate Program in Computer Science & Engineering

The Ohio State University

2020

Thesis Committee

Dr. D.K. Panda, Advisor

Dr. Radu Teodorescu

Dr. Hari Subramoni

Abstract

The Message Passing Interface (MPI) is a popular parallel programming model for developing parallel scientific applications. Collective operations defined in the MPI standard offer a convenient abstraction to implement group communication operations. Owing to their ease of use and performance portability, collective operations are used across various scientific domains. The MVAPICH2 library implements the collective operations using various algorithms. Each algorithm performs differently based on various factors such as message size, system size, CPU type, interconnect type, topology, etc. Thus, a question arises from the implementer's perspective--how can MVAPICH2 be optimized so that it utilizes the best algorithm for any combination of the factors mentioned above? To address this problem, we have created a "tuning" framework---a set of programs that compare the performance of various algorithms by varying the above factors and selects the best one for a given combination. Experimental results with the Tuning Framework show performance improvements of up to 79%. In addition to the Tuning Framework, this work introduces a Regression Framework which simplifies the understanding of change in performance between versions of MVAPICH2.

Acknowledgements

I would like to thank Dr. Panda for taking me in to his research group where I've had the chance to learn so much and meet some truly incredible people.

I would like to acknowledge Dr. Radu Teodorescu for being on my committee.

Dr. Hari Subramoni for answering my endless questions about MVAPICH2 and being a great mentor.

Quentin Anthony for QA testing my code and more importantly, being a great friend.

Dr. Jahanzeb Hashmi for his advice and mentorship.

Mamzi Bayatpour for helping me when I struggled and sharing laughter with me.

Table of Contents

Chapter 1: Introduction

In this chapter, we introduce the MPI parallel programming model and the collective operations within it. Then, we introduce collective "tuning" and regression.

Scientific applications in numerous domains--for example, weather forecasting and nuclear simulations--require a large amount of computing resources to be able to have meaningful results. Supercomputing clusters have been designed and created to meet this demand. However, the design of a supercomputer is much different to a desktop computer and thus is programmed differently as well. The large-scale architecture--some of the top supercomputers in the world having hundreds of thousands of CPUs within--demands a new and standardized programming model that is easy to understand yet powerful enough to be able to handle the processing and transfer of data between up to hundreds of thousands of CPUs, each running many processes.

The Message Passing Interface (MPI) is such a model. It defines two different sets of communication operations that allow the scientific programmer to transfer data between processes. The first are point-to-point operations in which data is transferred from one process to another, and the second are collective operations which involve the transfer of data between more than two processes.

This work focuses on the collective operations. There exist a number of them, including MPI_Allgather, MPI_Allreduce, MPI_Alltoall, MPI_Bcast, MPI_Gather,

MPI_Reduce, and MPI_Scatter. The communication schemes of MPI_Bcast, MPI_Scatter, MPI_Reduce, and MPI_Gather can be seen in Figure 1.

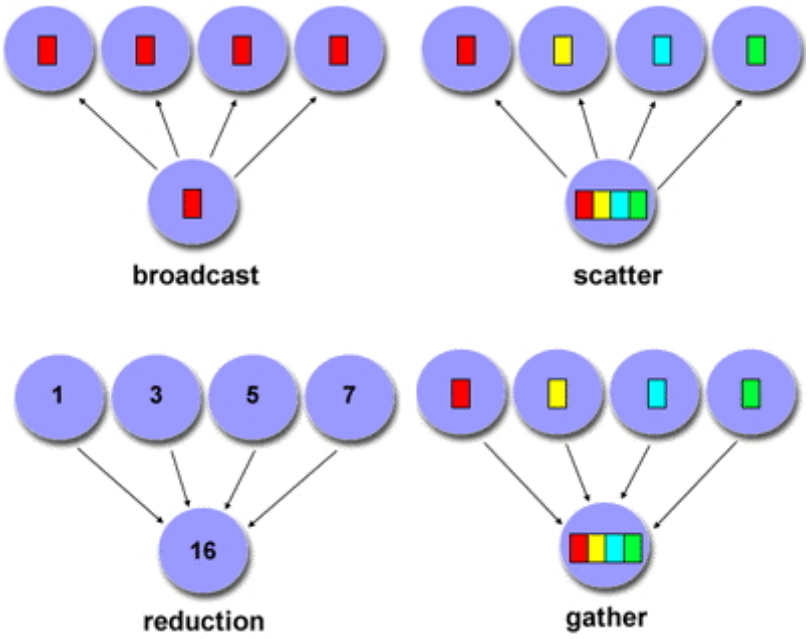


Figure 1: Communication Schemes for selected MPI Collectives

In Figure 1 [1], the color of each rectangle represents an arbitrary piece of data which can be thought of as one or more elements in a buffer. A buffer is simply an array whose data is to be sent or an array whose space is meant for data received. A communicator is a logical representation of a group of processes.

An MPI_Broadcast will take the entire buffer of one process and send it to the receive buffer of each of the rest of the processes in the communicator. An MPI_Scatter will take a buffer in one process, split it, and send each resulting piece of data from the split to the rest of the processes in the communicator. An MPI_Gather is an MPI_Scatter

but in reverse. A call to MPI_Reduce is similar to an MPI_Gather, except it will apply a reduction operation to each of the elements received. Some reduction operations are: MPI_MAX, MPI_MIN, MPI_SUM, and MPI_PROD to find the maximum element, minimum element, the sum of all elements, and the product of all elements, respectively. In the figure, MPI_SUM was the reduction operation.

The MVAPICH2 library [10] provides an API to the collective operations. It implements them using calls to the point-to-point operations such as MPI_Send, MPI_Recv, MPI_Isend, MPI_Irecv, and others. The manner in which each collective is implemented with these functions is up to the developer of the library and from now on is referred to as the algorithm.

Algorithms can vary in design, such as a whether they are flat or two-level. Two-level designs distinguish between inter-node and intra-node transfers of data. This distinction is made because inter-node transfers go over the network (more costly) while intra-node transfers are done via shared-memory (less costly) within MVAPICH2. The reader is encouraged to peruse the design of some two-level collective algorithms in [11], [12], [13], [14].

Each collective operation within MVAPICH2 is implemented with several different algorithms. However, because the user of the MPI library does not care about which algorithms are used, instead only caring that the library works and performs optimally on their system, this leads to an issue that arises from the perspective of the implementer--they must somehow select which algorithm to run for each of the operations. This selection process is known as "tuning" and involves benchmarking each algorithm in order to find the one with the lowest latency.

Once a cluster has been tuned, there is a need to verify the performance has improved. To do this, before-and-after testing (dubbed "Regression") is done after every tuning patch is taken in to the MVAPICH2 code. In a regression, the percent change (before and after) of every combination of nodes, processes per node (PPN), and message size are reported.

Chapter 2: Current Approach and Limitations

In this chapter, we detail the current approach to tuning the collective operations within MVAPICH2 and its limitations.

The current approach to tuning requires running a 2000-line C program which was converted from a bash script that did the same thing. To do the conversion, each line in the bash script was wrapped with this macro:

```
#define RUN_COMM(...)
do {
    comm [0] = 0;
    snprintf (__VA_ARGS__);
    system (comm);
} while(0)
```

It calls the "system" function in order to execute the bash commands from C. This is *highly* unmaintainable. In addition, the C program is error-prone and difficult to configure for new supercomputing clusters. The program is also not fault-tolerant--it requires a full restart in the event of an error. For tuning jobs that are up to 8 hours long, this is an extremely inefficient use of time and computing resources. Lastly, the benchmark data generated is saved to the filesystem in an unorganized manner. After running, the user deletes this data. If the data is needed again, the benchmark data must be rerun.

For regression, the current approach is to use a framework written in bash. The framework works, but it has had several complaints in the past regarding usability. In addition, it does not provide "rules"-based sanity checking. For example, MPI_Allreduce should always have less latency than MPI_Reduce + MPI_Bcast combined. This is because semantically, they are the same operation: an MPI_Allreduce will have the same effect as

an MPI_Reduce followed by MPI_Broadcast. In order for MPI_Allreduce to be useful, it must have better performance than the latter combination. There are several such rules for the rest of the collective operations.

Chapter 3: Problem Statement

This work poses following two questions:

- How can a Tuning Framework be designed for maintainability, fault-tolerance, and have algorithm data permanently stored?
- How can a Regression Framework be designed to also permanently store data, with an emphasis on ease-of-use?

To answer these, we use Python for maintainability and ease-of-use. We use MongoDB for permanent algorithm and regression data storage as well as fault-tolerance. In the event a job fails, all data is saved.

Chapter 4: Background

In this Chapter, we provide an overview of works that the Tuning and Regression Framework are built upon. In addition, we explain the structure of the "tuning tables" the Tuning Framework generates and that the MVAPICH2 code reads in.

4.1 MPI

The Message Passing Interface (MPI) is a standard for the message passing programming model. The aim of the MPI standard is to define a flexible, easy-to-use, and portable interface to parallel programming through Message Passing. The Message Passing Interface is authored by the MPI Forum, consisting of over 40 participating member organizations including researchers, venders, and users [1]. MPI has been the dominant programming model in the High Performance Computing space for over 20 years.

4.2 MVAPICH2

Based on the MPI 3.1 standard, MVAPICH2 provides high performance, scalability, and fault tolerance to large-scale computing systems using Infiniband, Omni-Path, Ethernet/iWARP and RoCE. More than 3,075 organizations consisting of National Laboratories, Universities, and companies in the Industry have registered as users of the MVAPICH2 software on the project's website [2]. The MVAPICH2 software runs on several top supercomputing systems on the TOP500 list [3], including 3rd, 10,649,600-core (Sunway TaihuLight) at National Supercomputing Center in Wuxi, China and 5th, 448,448 cores (Frontera) at Texas Advanced Center for Computing.

4.3 OSU MicroBenchmarks

The OSU MicroBenchmarks suite provides a variety of tests to evaluate MPI and PGAS libraries for CPUs and GPUs. The MPI tests fall under 5 different categories: Point-to-Point Benchmarks, Collective Benchmarks, Non-Blocking Collective Benchmarks, One-Sided Benchmarks, and Startup Benchmarks. This work heavily relies on the Collective Benchmarks. Within this category, tests exist for MPI_Allgather, MPI_Allreduce, MPI_Alltoall, MPI_Bcast, MPI_Gather, MPI_Reduce, and MPI_Scatter [3].

4.4 Tuning Tables Within MVAPICH2

MVAPICH2 must be aware of the best algorithm for the given job configuration in order to select it. Once generated with the Tuning Framework, the best algorithms are stored within header files in the MVAPICH2 source code. The contents of the header files are known as "tuning tables" and have a list of message sizes with their corresponding algorithms. Each header file has a designated number of processes per node (PPN), and all tuning tables within the file correspond to a number of nodes. A sample table is found in Figure 2. The figure shows the 1 Node table within the 16 PPN file for the Frontera supercomputer at TACC. The table is as follows: a macro representing the tuning table is defined in line 1. The name of the macro follows the convention CHANNEL_ARCHITECTURE_#PPN. In line 3, the number of processes is set (here it is 16, for 1 Node * 16 PPN). In line 5, whether the algorithm at a particular message size is two-level is set. In lines 6 and 28, the number of message sizes is set. Here it is 19 for all the powers of two between 4 bytes and 1MB. The top section corresponds to the inter-node

functions for two-level algorithms, the bottom section corresponds to the intra-node functions for two-level algorithms. If the message size is set to be flat (not two-level), only the inter-node algorithm is selected). Within each section is a set of mappings from each message size to the best algorithm for that size.

```
#define GEN2_CMA__MV2_ARCH_INTEL_XEON_E5_2620_V4_2S_16__16PPN {  \
    {   \
    16, \
    0,  \
    {1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  \
    19, \
    {   \
    {4, &MPIR_Allreduce_pt2pt_rs_MV2},  \
    {8, &MPIR_Allreduce_pt2pt_rd_MV2},  \
    {16, &MPIR_Allreduce_pt2pt_rd_MV2}, \
    {32, &MPIR_Allreduce_pt2pt_rd_MV2}, \
    {64, &MPIR_Allreduce_pt2pt_rd_MV2}, \
    {128, &MPIR_Allreduce_pt2pt_rd_MV2},    \
    {256, &MPIR_Allreduce_pt2pt_rd_MV2},    \
    {512, &MPIR_Allreduce_pt2pt_rd_MV2},    \
    {1024, &MPIR_Allreduce_pt2pt_rd_MV2},   \
    {2048, &MPIR_Allreduce_pt2pt_rd_MV2},   \
    {4096, &MPIR_Allreduce_pt2pt_rs_MV2},   \
    {8192, &MPIR_Allreduce_pt2pt_rs_MV2},   \
    {16384, &MPIR_Allreduce_pt2pt_rs_MV2},  \
    {32768, &MPIR_Allreduce_pt2pt_rs_MV2},  \
    {65536, &MPIR_Allreduce_pt2pt_rs_MV2},  \
    {131072, &MPIR_Allreduce_pt2pt_rs_MV2}, \
    {262144, &MPIR_Allreduce_pt2pt_rs_MV2}, \
    {524288, &MPIR_Allreduce_pt2pt_rs_MV2}, \
    {1048576, &MPIR_Allreduce_pt2pt_rs_MV2},    \
    },  \
    19, \
    {   \
    {4, &MPIR_Allreduce_reduce_shmem_MV2},  \
    {8, &MPIR_Allreduce_reduce_shmem_MV2},  \
    {16, &MPIR_Allreduce_reduce_shmem_MV2}, \
    {32, &MPIR_Allreduce_reduce_shmem_MV2}, \
    {64, &MPIR_Allreduce_reduce_shmem_MV2}, \
    {128, &MPIR_Allreduce_reduce_shmem_MV2},    \
    {256, &MPIR_Allreduce_reduce_shmem_MV2},    \
    {512, &MPIR_Allreduce_reduce_shmem_MV2},    \
    {1024, &MPIR_Allreduce_reduce_p2p_MV2}, \
    {2048, &MPIR_Allreduce_reduce_p2p_MV2}, \
    {4096, &MPIR_Allreduce_reduce_p2p_MV2}, \
    {8192, &MPIR_Allreduce_reduce_p2p_MV2}, \
    {16384, &MPIR_Allreduce_reduce_p2p_MV2},    \
    {32768, &MPIR_Allreduce_reduce_p2p_MV2},    \
    {65536, &MPIR_Allreduce_reduce_p2p_MV2},    \
    {131072, &MPIR_Allreduce_reduce_p2p_MV2},   \
    {262144, &MPIR_Allreduce_reduce_p2p_MV2},   \
    {524288, &MPIR_Allreduce_reduce_p2p_MV2},   \
    {1048576, &MPIR_Allreduce_reduce_p2p_MV2},  \
    }   \
    },  \
```

Figure 2: Sample Allreduce Tuning Table

4.5 Python

Python is an interpreted, high-level, object-oriented programming language [5] developed by Guido Van Rossum in 1991. Python has become a popular language due to its focus on readability and simplicity, and an abundance of available packages. Python is the primary language used in this work.

4.6 Matplotlib

Matplotlib is an objected-oriented API [6] for the Python programming language to easily create charts, tables, and other graphics to represent data. The Matplotlib library is used in this work as part of the Regression Framework.

4.7 Reportlab

Reportlab is a Python library for dynamically generating PDF documents. The Reportlab software provides a low-level API for drawing lines and shapes onto a page. It also provides a high-level API called Platypus that provides a workflow for generating documents. To use Platypus, the user must create Flowable objects and append them to a list which is passed to the Document object. The method Document.build is then called to generate the PDF output file. Flowables can be pre-defined classes such as Paragraph and Image, or they can be user-defined to behave in a custom manner.

Chapter 5: Design

In this chapter, we discuss the architecture of the Tuning and Regression Frameworks. Then, we detail the code design. Both Frameworks were written with the Object-Oriented Programming methodology, so a discussion of several classes critical to the Frameworks is presented.

5.1 Tuning and Regression Framework Architecture

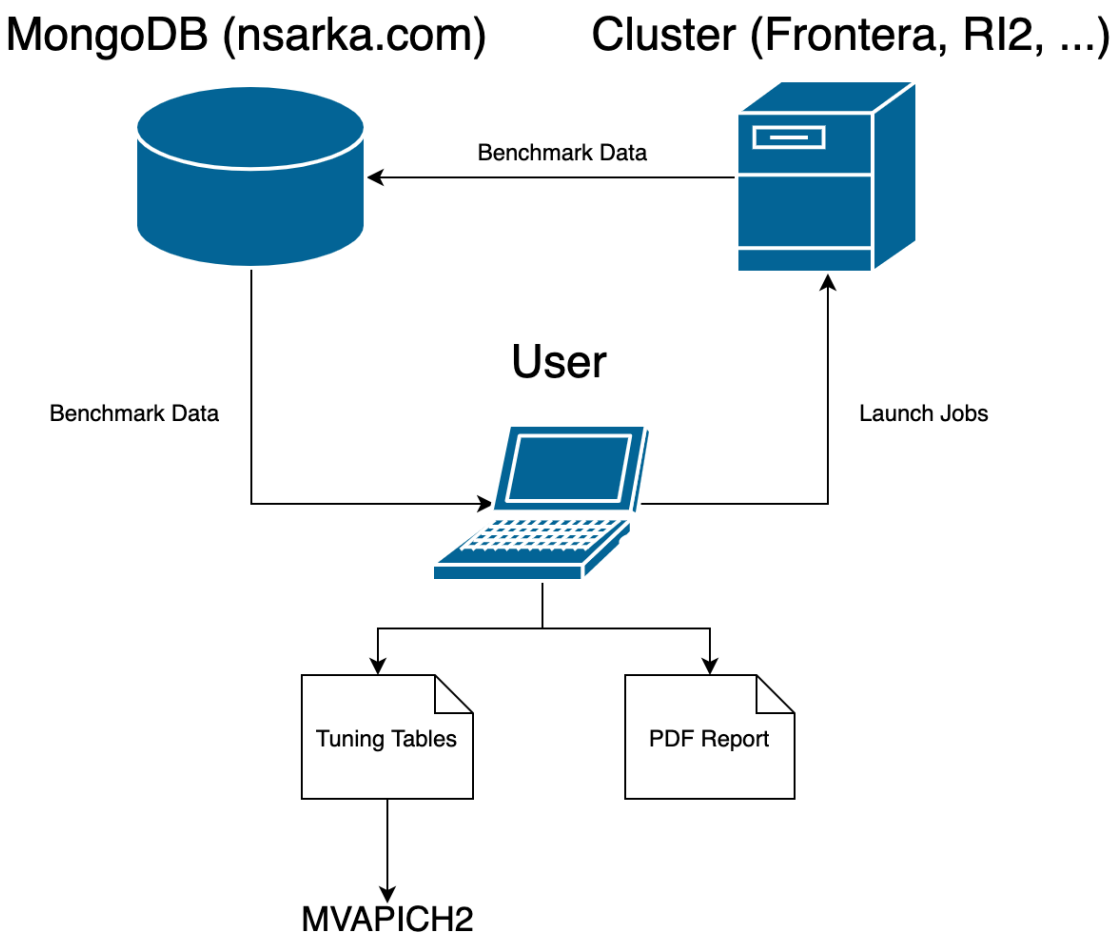


Figure 3: Tuning and Regression Framework Architecture

The user will define and launch tuning jobs that run on a supercomputing cluster. As the jobs run, benchmark data gets saved to a MongoDB instance that is running on our personal website. After all jobs complete, the user generates tuning tables and/or PDF reports based on the benchmark data. Tuning tables are then added as a patch to MVAPICH2 and PDF reports are distributed among other developers for viewing. Figure 3 illustrates this process.

5.2 Code Design

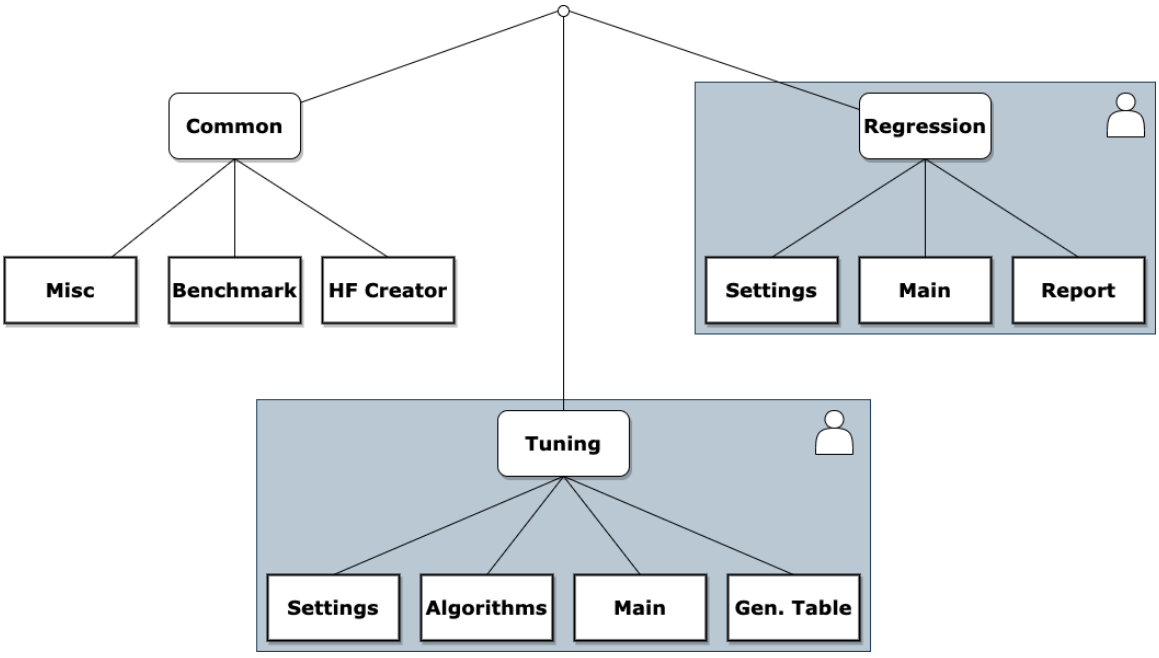


Figure 4: Code Design of the Tuning and Regression Framework

The code design of the Tuning and Regression Framework can be seen in Figure 4. The figure is split into three sections corresponding to the three folders within the source code. Within the two gray sections are files the user interacts with.

The section that does not have a gray box, common, contains classes that are designed to be reusable for any script that requires the running of OSU MicroBenchmarks with MVAPICH2. These classes are highly flexible and we recommend any future scripts whose generated data must be saved to be written with them. The Tuning and Regression Frameworks can serve as demonstrations to this flexibility: the core logic for tuning is in a file, tuning.py, that is only 67 lines long. This is a massive improvement to the old framework for tuning, a C program at 2000 lines of code.

Next, we detail the Benchmark and HostfileCreator classes. After these classes, we detail the Settings.ini files of the Tuning and Regression Frameworks to showcase the usability improvements over the old tuning scripts.

5.3 Benchmark Class

The software architecture rests heavily on the Benchmark class found within the Common folder. The class is a high-level scripting interface to an OSU Microbenchmark. With it, a developer of MVAPICH2 can easily specify a Benchmark to run and also read/save it into the database. In addition, the +, -, /, *, <, >, and == operators are all overloaded within the class. As a result, the developer may add the data of two Benchmarks, subtract the data, and so on. The comparison operators <, >, and ==, check to see whether each message size within the benchmark are less then 5%, greater than 5%, or within 5% of the other. If the condition holds, the expression returns True. During the

development of the Regression Framework, these operators proved extremely useful, especially while calculating whether or not a set of benchmarks passed their respective rules, which are detailed later.

The Benchmark class makes it easy to interact with the database. The Benchmark.read_db() method returns a list of Benchmarks that have the same description of the benchmark that called the method. In other words, if the developer creates a new Benchmark object and specifies a CPU architecture, Benchmark.read_db() will return a list of Benchmark objects (and their corresponding data within them) from the database.

5.4 HostfileCreator Class

The HostfileCreator class allows the automatic creation of a hostfile either by passing in a list of hostnames or by setting the job scheduler that the cluster uses. Within the script, the developer will generally use the following two lines:

```
hf_creator = HostfileCreator(job_name=job_name, hostfile_dir=hostfile_folder, scheduler=scheduler)

hostfile = hf_creator.create(ppn, nodes)
```

The user now will have a path to a hostfile stored in the hostfile variable.

5.5 Settings.ini

The Settings file enables the division of all of the tuning configurations into user-specified jobs. Previously, a complicated mix of command line arguments and files were used. With the new Settings.ini file, the user defines a job that has the number of nodes, processes per node, benchmarks, environment variables, and the channel. The user then

submits the main.py script with the job name (enclosed in brackets) and the configuration will be applied to the job. A sample job configuration is below in Figure 5.

```
[ri2-4-nodes]
Nodes: 1, 2, 4
PPN: 1, 2, 4, 8
Channel: gen2_cma
Benchmarks:collective/osu_allreduce,collective/osu_bcast,collectiv
e/osu_gather,collective/osu_reduce,collective/osu_scatter
Envs: MV2_USE_RDMA_CM=0
```

Figure 5: Sample Job Configuration

With this sample job configuration, the user should submit a job with 4 nodes (because the job will expect up to 4 nodes based on the Nodes section). The job will run every benchmark for every combination of nodes and processes per node within the configuration, excluding 1 node 1 process per node. Each run will have the environment variables within the Envs section set. The channel can be set to gen2, gen2_cma, or psm.

To submit this job with Slurm, the user can run `salloc -N 4 ./submit.sh ri2-4-nodes`. The submit.sh script is a wrapper around `python main.py` that activates the conda environment. If Miniconda is not used, using submit.sh is not necessary.

Next, we detail the design of the Tuning and Regression core logic.

5.6 Tuning Logic

As mentioned earlier, the core logic for tuning is only 67 lines long with the help of the Benchmark class. It is one function, Tuning_Take_Numbers, which does the following:

1. Read Setting.ini for the job configuration the user passed.
2. Open algs.json, a file containing all the algorithms for each collective and the environment variables necessary to force them.
3. Instantiate a Hostfile Creator
4. Create a list data structure, and, for each one of the nodes, ppn, and benchmarks set, append a new Benchmark instance to the list
5. Iterate over the list calling Benchmark.run() and Benchmark.save()

5.7 Regression Logic

The regression logic follows the same workflow as the tuning logic, except that it creates two of each Benchmark in the configuration. One is for the old version and the other is for the new version. Each is run and saved to the database.

Next, we detail the Tuning Table Generator of the Tuning Framework and also the Report generator of the Regression Framework

5.8 Tuning Table Generator

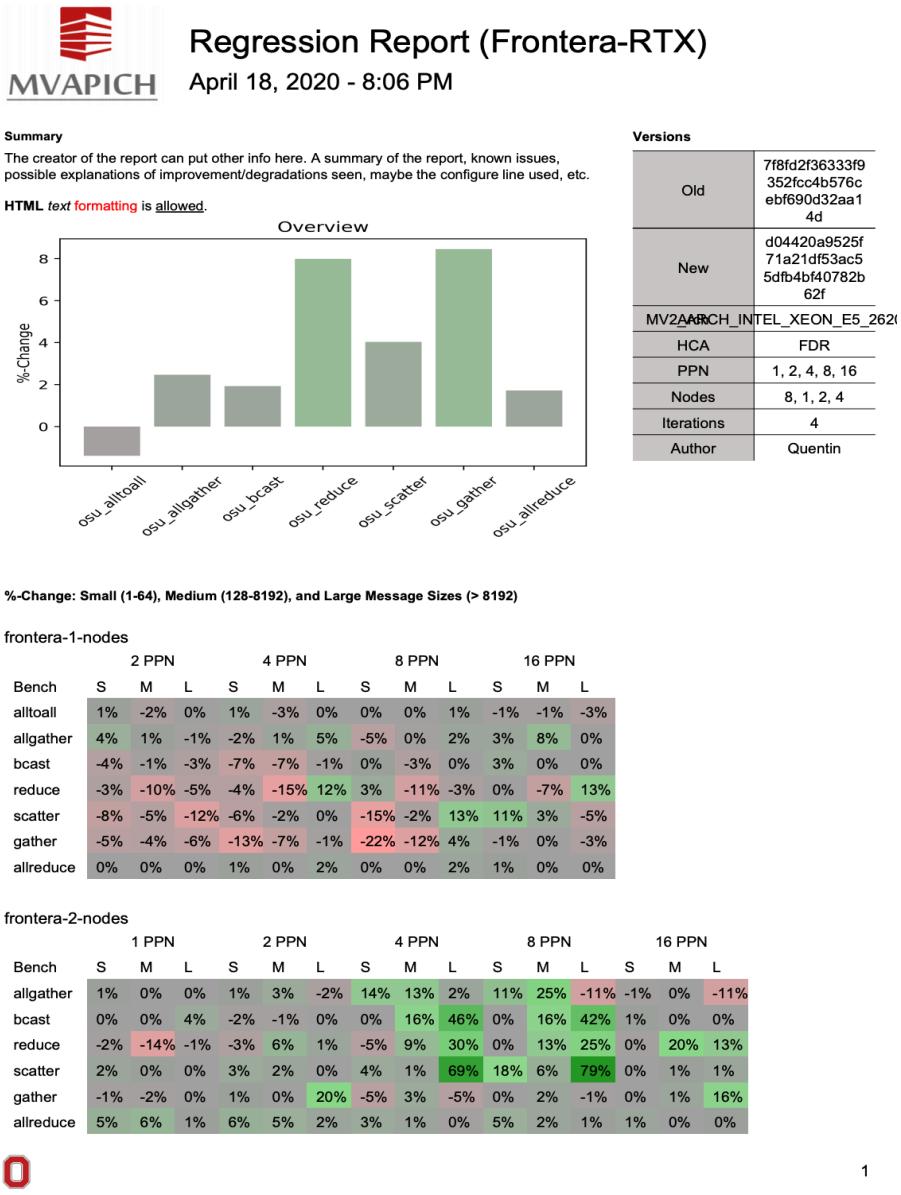
The file for the tuning table generator is generate_table.py. It does not make use of the settings.ini configuration file, instead opting to read in command line arguments for the

benchmarks, nodes, processes per node, and so on. The main reason for this is that the settings.ini file was meant as a way for users to break up their jobs. The Table Generator will create tuning tables based on the aggregate of the jobs that the user submits. So, the settings.ini file is not used for reading in parameters of jobs. It is used, however, for reading in the database configuration.

Once the benchmarks, nodes, processes per node, and so on are read in, a list of Benchmarks is created with these parameters in combination with the environment variables needed to force the different collective algorithms within MVAPICH2. Each benchmark then calls read_db(). Whichever algorithm had the lowest latency across all message sizes is the algorithm that gets selected. The variance of the latency of each message size was considered, but during the development of the Tuning Framework it was found that simply finding the minimum was sufficient to generate the best tables.

5.9 Report Creation

With the Regression Framework comes a tool for generating a PDF report of the regression results. The first page of a sample report can be seen in Figure 5.



MVAPICH

Regression Report (Frontera-RTX)

April 18, 2020 - 8:06 PM

Summary

The creator of the report can put other info here. A summary of the report, known issues, possible explanations of improvement/degradations seen, maybe the configure line used, etc.

HTML *text* formatting is allowed.

Versions

Old	7f8fd2f36333f9352fcc4b576cebf690d32aa14d
New	d04420a9525f71a21df53ac55dfb4bf40782b62f
MV2_ARCH	INTEL_XEON_E5_262(
HCA	FDR
PPN	1, 2, 4, 8, 16
Nodes	8, 1, 2, 4
Iterations	4
Author	Quentin

%-Change: Small (1-64), Medium (128-8192), and Large Message Sizes (> 8192)

frontera-1-nodes

	2 PPN			4 PPN			8 PPN			16 PPN		
Bench	S	M	L	S	M	L	S	M	L	S	M	L
alltoall	1%	-2%	0%	1%	-3%	0%	0%	0%	1%	-1%	-1%	-3%
allgather	4%	1%	-1%	-2%	1%	5%	-5%	0%	2%	3%	8%	0%
bcast	-4%	-1%	-3%	-7%	-7%	-1%	0%	-3%	0%	3%	0%	0%
reduce	-3%	-10%	-5%	-4%	-15%	12%	3%	-11%	-3%	0%	-7%	13%
scatter	-8%	-5%	-12%	-6%	-2%	0%	-15%	-2%	13%	11%	3%	-5%
gather	-5%	-4%	-6%	-13%	-7%	-1%	-22%	-12%	4%	-1%	0%	-3%
allreduce	0%	0%	0%	1%	0%	2%	0%	0%	2%	1%	0%	0%

frontera-2-nodes

	1 PPN			2 PPN			4 PPN			8 PPN			16 PPN		
Bench	S	M	L	S	M	L	S	M	L	S	M	L	S	M	L
allgather	1%	0%	0%	1%	3%	-2%	14%	13%	2%	11%	25%	-11%	-1%	0%	-11%
bcast	0%	0%	4%	-2%	-1%	0%	0%	16%	46%	0%	16%	42%	1%	0%	0%
reduce	-2%	-14%	-1%	-3%	6%	1%	-5%	9%	30%	0%	13%	25%	0%	20%	13%
scatter	2%	0%	0%	3%	2%	0%	4%	1%	69%	18%	6%	79%	0%	1%	1%
gather	-1%	-2%	0%	1%	0%	20%	-5%	3%	-5%	0%	2%	-1%	0%	1%	16%
allreduce	5%	6%	1%	6%	5%	2%	3%	1%	0%	5%	2%	1%	1%	0%	0%

1

Figure 6: Sample Report

The report generation program can be launched with the command: "python main.py --report <job_name_1> <job_name_2> ... <job_name_n>" to create a report for

the specified jobs within settings.ini. ReportLab is used for generating the PDF. A sample report is shown in Figure 6 that shows the regression between the MVAPICH2 master branch (as of April 18th, 2020) compared to MVAPICH2 with tuning tables for Frontera added. The "Overview" chart is the highest-level representation of the regression data. It shows the average percent change for each benchmark run taking into account all combinations of nodes and PPN.

The "Versions" table shows the CPU architecture, interconnect, nodes used, PPN used, the number of iterations each benchmark is run for, and the author of the report. The table also shows the commit hash for the old and new versions of the MVAPICH2 code. It is important that the user uses a commit hash instead of the name of a branch (for example, 'master', or 'master-x') because the commit a branch points to is constantly changing as the code is developed.

The "%-Change" table illustrates the average percent change in message sizes grouped by small (1 to 64 bytes), medium (128-8192 bytes), and large (over 8192 bytes). Each table corresponds to a job with some value for the number of nodes. Each PPN run in the job gets its own S, M, L grouping. This table format allows for fast understanding of the change in performance of the MVAPICH2 code.

Figure 7 shows the second page of a sample report.

frontera-4-nodes

	1 PPN			2 PPN			4 PPN			8 PPN			16 PPN		
Bench	S	M	L	S	M	L	S	M	L	S	M	L	S	M	L
allgather	0%	0%	0%	0%	1%	5%	13%	14%	6%	0%	3%	4%	3%	2%	0%
bcast	-2%	0%	1%	1%	1%	0%	2%	24%	45%	-3%	0%	4%	0%	0%	0%
reduce	1%	3%	6%	0%	-1%	10%	0%	16%	23%	1%	21%	19%	0%	0%	-6%
scatter	1%	0%	0%	2%	0%	0%	25%	9%	57%	10%	9%	0%	1%	1%	1%
gather	0%	0%	19%	0%	4%	33%	10%	12%	-3%	0%	0%	15%	-1%	3%	17%
allreduce	1%	2%	0%	1%	0%	0%	2%	1%	0%	0%	0%	0%	1%	1%	-1%

frontera-8-nodes

	1 PPN			2 PPN			4 PPN			8 PPN			16 PPN		
Bench	S	M	L	S	M	L	S	M	L	S	M	L	S	M	L
allgather	0%	1%	2%	1%	5%	12%	0%	7%	8%	1%	4%	3%	1%	1%	0%
bcast	0%	0%	6%	-1%	0%	4%	1%	2%	3%	-2%	1%	0%	-3%	-1%	0%
reduce	0%	2%	5%	0%	0%	16%	0%	21%	24%	0%	0%	0%	0%	0%	7%
scatter	1%	0%	0%	0%	0%	0%	19%	14%	0%	0%	0%	0%	2%	0%	0%
gather	1%	0%	22%	0%	0%	16%	6%	4%	18%	0%	0%	10%	-1%	3%	12%
allreduce	0%	1%	0%	0%	4%	1%	-1%	0%	0%	2%	2%	1%	1%	5%	1%

Rules

Applicable rules for the jobs within this report are listed below. For a benchmark to be less than, equal to, or greater than another benchmark, it must be < 5%, within 5%, or > 5% for every message size.

frontera-1-nodes

	2 PPN	4 PPN	8 PPN	16 PPN
Bcast < Gather	PASS	PASS	PASS	FAIL
Gather < Allgather	FAIL	PASS	PASS	PASS
Allgather < Alltoall	FAIL	FAIL	FAIL	FAIL
Allgather < Gather + Bcast	FAIL	FAIL	FAIL	FAIL
Reduce < Allreduce	FAIL	FAIL	FAIL	FAIL
Allreduce < Gather	FAIL	FAIL	FAIL	FAIL
Allreduce < Reduce + Bcast	FAIL	FAIL	FAIL	FAIL
Gather = Scatter	FAIL	FAIL	FAIL	PASS

frontera-2-nodes

	1 PPN	2 PPN	4 PPN	8 PPN	16 PPN
Bcast < Gather	FAIL	FAIL	FAIL	FAIL	FAIL
Gather < Allgather	FAIL	PASS	PASS	PASS	PASS
Allgather < Gather + Bcast	PASS	FAIL	FAIL	FAIL	FAIL
Reduce < Allreduce	FAIL	PASS	FAIL	FAIL	FAIL
Allreduce < Gather	FAIL	FAIL	FAIL	FAIL	FAIL
Allreduce < Reduce + Bcast	PASS	FAIL	FAIL	FAIL	FAIL
Gather = Scatter	FAIL	FAIL	FAIL	FAIL	PASS

Figure 7: Sample Report Page Two

Within the second page is a continuation of the %-Change tables from the first. After the %-Change tables, the Rules section is shown. A "Rule" is one of the listings in the left column of the Rules tables and serve to check the "sanity" of the collectives. For example, the MPI_Allreduce function can be implemented with MPI_Reduce followed by MPI_Bcast. Thus, the MPI_Allreduce function should have better performance than the other two combined, or else the user will not use it. Rules will only show up if data for applicable collectives exist. If either one of MPI_Allreduce, MPI_Reduce, or MPI_Bcast was not tested, the rule will not appear in the report.

Lastly, the report generates graphs with Matplotlib of each job configuration run. An example graph is shown in Figure 8. The rest of the graphs within the report are omitted for brevity.

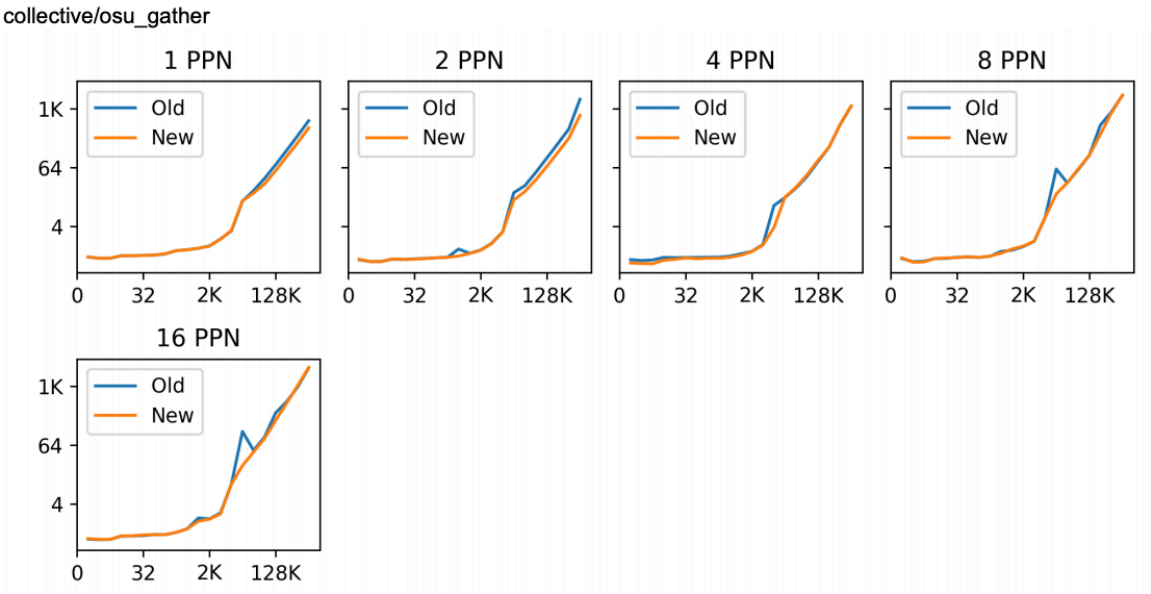


Figure 8: Sample Report Graphs. osu_gather at 4 Nodes

Chapter 6: Experimental Results

In this chapter, we discuss the experimental results of the Tuning and Regression Frameworks. In order to demonstrate their effectiveness, we ran the Tuning Framework on the Frontera Supercomputer at Texas Advanced Center for Computing (TACC). On Frontera, we tuned the RTX nodes containing Intel Xeon E5 2620 with FDR InfiniBand. We then ran the Regression Framework and created a report detailing the before and after effects.

Figures 6, 7, and 8 illustrate the benefits of tuning with the Tuning Framework in addition to showcasing the Regression Framework. From the %-Change tables, it is plainly visible that there are many cases of drastically improved performance in the collectives. MPI_Scatter with 4 nodes show performance benefits of up to 79%. MPI_Bcast shows a performance increase of up to 46% in a 2 node job with 4 PPN. In the Overview chart, it can be seen that MPI_Reduce and MPI_Gather increased by 8% and 9% respectively. While this may seem like a small amount at first, the reader must take into account that this figure is an average between benchmarks with all combinations of nodes and PPN. In many cases, the latency of the benchmark was drastically reduced. In other cases, the default selected algorithm had the optimal performance. This means that there is no change for such cases. The cases with no change bring down the average. So, to say that a collective improved as a whole by a factor of 8% is formidable.

Cases with decreased performance, while rare, do still exist. This can be attributed to variation within the job. Factors such as stray processes and operating system scheduling may take part in explaining decreases in performance. To alleviate this, tuning can be rerun on a different set of nodes.

The rules section shows an under-developed aspect of the MVAPICH2 library. While there are passes, there are many failures in the rules. The Regression Framework, to the best of our knowledge, is the first work to be developed internally that tests collective rules on the MVAPICH2 library. Now that there is a tool that works seamlessly with MVAPICH2 and OSU Microbenchmarks to test rules, this aspect may be more easily developed further.

Chapter 7: Conclusion and Future Work

7.1 Conclusion

The Tuning Framework allows the developer of MVAPICH2 to find the best algorithms for the MPI collectives among numerous factors including the system CPU architecture, system interconnect, number of nodes, number of processes per node, and message size. With the use of Python and the addition of a database to the Framework, the process of tuning is greatly reduced in terms of time spent of the user and also in terms of CPU resources. In the case there was a problem, or that the data is needed in the future, data no longer needs to be retaken. It can simply be retrieved from the database. At the time of this writing, the Tuning Framework is being used for tuning collectives on Frontera as well as an AMD Rome cluster in Austin, TX managed in-house by AMD. These works will be a part of the next MVAPICH2 release: version v2.3.4.

The Regression Framework allows the developer to compare the performance between versions of MVAPICH2 and generate a report detailing at a high level not only the performance difference, but also "rules" that can provide a sanity check of the collectives.

7.2 Future Work

We plan to add support for tuning the point-to-point operations within the MVAPICH2 library. This involves determining the Eager Threshold--the point which the transfer protocol switches from Eager to Rendezvous. The process for determining the eager threshold is relatively straight-forward: a set of environment variables is used to force different values of the threshold. A comma-separated-value output file is generated after

running. Each threshold value gets a column within the output file. Each row of the output file corresponds to a message size. It is up to the user to determine the optimal eager threshold by examination of the file.

A similar process is performed for determining the "VBUF" size. This size is the length of the buffer used for sending and receiving data with the eager protocol. Both values, the eager threshold and the vbuf size, get saved into the MVAPICH2 code as a patch.

It is worth noting that there exists a script for tuning the point-to-point operations already. However, we plan to add point-to-point tuning anyways in an effort to unify all types of tuning for a cluster. The workflow will be similar to tuning collectives (using the settings.ini configuration file to split jobs) and thus the user will not need to learn how to use a new script.

We also plan to explore using Machine Learning (ML) techniques in order to estimate the optimal algorithms for a given system and job configuration. Most frameworks for machine learning (such as PyTorch [8]) use Python, so experimenting with ML is a natural next step for this work.

References

[1] Blaise, B. (n.d.). *Message passing interface (MPI) Tutorial*. Lawrence Livermore National Laboratory. https://computing.llnl.gov/tutorials/mpi/

[2] Network Based Computing Laboratory. (2019, March 1). *MVAPICH2 Userguide*. MVAPICH. https://mvapich.cse.ohio-state.edu/static/media/mvapich/mvapich2-2.3.1-userguide.html#x1-20001

[3] TOP500. (n.d.). *Top 500 Supercomputing Sites*. Home | TOP500 Supercomputer Sites. https://www.top500.org/

[4] Network Based Computing Laboratory. (2019). *OSU MicroBenchmarks*. MVAPICH. https://mvapich.cse.ohio-state.edu/benchmarks/

[5] ReportLab. (2020). *ReportLab Userguide*. ReportLab - Content to PDF Solutions. https://www.reportlab.com/docs/reportlab-userguide.pdf

[6] Python Software Foundation. (2020). *General Python FAQ — Python 3.8.2 documentation*. 3.8.2 Documentation. https://docs.python.org/3/faq/general.html#what-is-python

[7] Matplotlib Development Team. (n.d.). *Usage guide — Matplotlib 3.2.0 documentation*. Matplotlib: Python plotting — Matplotlib 3.2.1 documentation. https://matplotlib.org/3.2.0/tutorials/introductory/usage.html#sphx-glr-tutorials-introductory-usage-py

[8] MongoDB, Inc. (n.d.). *MongoDB*. MongoDB. https://www.mongodb.com/

[9] Facebook. (n.d.). PyTorch. https://pytorch.org/

[10] "MVAPICH2," NOWLAB, OSU, [Online]. Available: https://mvapich.cse.ohiostate.edu/.

[11] Scalable Reduction Collectives with Data Partitioning-based Multi-Leader Design
M. Bayatpour, S. Chakraborty, H. Subramoni, X. Lu, D. Panda
SuperComputing 2017,
Nov 2017.

[12] Designing Multi-Leader-Based Allgather Algorithms for Multi-Core Clusters
K. Kandalla, H. Subramoni, G. Santhanaraman, D. Panda
International Workshop on Communication Architecture for Clusters (CAC'09),
May 2009.

[13] Scaling Alltoall Collective on Multi-core Systems
R. Kumar, A. Mamidala, D. Panda
International Workshop on Communication Architecture for Clusters,
Apr 2008.

[14] SALaR: Scalable and Adaptive Designs for Large Message Reduction Collectives
M. Bayatpour, J. Hashmi, S. Chakraborty, H. Subramoni, P. Kousha, D. Panda
IEEE Cluster 2018,
Sep 2018.